A Putting Functional Programming to Work
Software Design and Verification using Domain Specific Languages

Introduction

Telecommunications systems have grown enormously in complexity since the bygone days of electromechanical exchanges. Today, telecoms networks consist of a wide variety of products—radio base stations, media gateways, radio network controllers, and so on—each a powerful computer system in its own right. These products communicate with each other via a plethora of complex protocols. Developing the software that controls these products, correctly and in time, is a challenging task indeed.

The signal processing that will soon be needed to provide ever-higher bandwidth to mobile broadband users is *extremely* computationally demanding. The constantly changing technology and continuous investments by operators mean that time-to-market is critically important for telecoms products—but at the same time the quality demands are dizzying: telecoms systems are expected to be running and available at least 99.999% of the time! Meeting all these requirements simultaneously is difficult. Thus, telecoms software is a fruitful source of interesting problems for researchers to address—problems whose solutions are of considerably wider interest. Ericsson's keen interest in software research has led to fruitful joint projects with our group, which form a key foundation for this proposal.

For example, consider the digital signal processing in a Radio Base Station. The task of the uplink signal processing is to take the radio signal from the air (echoes, interference and all), convert it from an analogue signal to a stream of digital measurements, and then transform it digitally into hundreds of data streams coming from the individual user equipment (telephones, laptops, etc). The data streams are of many different types, of different bandwidth—and the configuration of data streams is recomputed every millisecond! The signal processing algorithms are programmed in C, but this is not C as we know it—the code is sprinkled with pragmas that control compiler optimisations, and loop bodies are written *taking into account* things such as that the intended target processor can perform four multiplications in one cycle. This makes the code *very* hardware dependent—changing to the next generation processor, or worse still, a different processor vendor, may ruin its performance.

The signal processing is so computationally demanding that it cannot be performed on just one DSP processor—it must be partitioned across many processing cores, each of which implements a composition of stream-to-stream functions. This partitioning is done early in the software development process, and the software for each core may even be developed by different teams. If one core turns out to be overloaded when the software is complete, then moving a part of its task to another is extremely difficult. Although in principle one need only move a few of the composed functions from one core to the other, in practice the individual functions are not apparent in the optimised code—so the code for each core must be rewritten and reoptimised. Since tests are performed at the interfaces between cores, even the test suite must be recreated to support this kind of change.

How might software development of this kind be eased? Evidently, the signal processing algorithms should be programmed at a higher level of abstraction—one that permits the kind of low-level control needed to obtain the necessary performance, but preserves the structure of the algorithms as compositions of stream functions, so that reconfiguration onto different hardware is possible from the same source code. Manual benchmarking and exploration of coding alternatives should be automated. Test cases should not be constructed by hand, but generated from a specification, in a way that can cope with reconfiguration. These are the goals of the DSL4DSP project, a part of this application.

Goals Our overall goal, in this project, is to develop methods and tools that can ease the specification, design, implementation, and verification of such challenging software. Our chief weapons for this task are *functional programming* in general, and *domain-specific embedded languages* in particular.

Domain specific languages (DSLs) are languages tailored to solving a specific task well, but without needing to be applicable outside the domain—such as make-files, for example. DSLs have seen very many applications in recent years, and offer the potential to provide automatic generation of optimised code, with fine control over low-level details where necessary, combined with a high-level and compositional overall approach. DSLs are particularly easy to experiment with when *embedded* in a host language—the DSL is provided as a library API, and reuses the host language's parser, type-checker, and other tools, thus dramatically reducing the implementation effort needed, and enabling more design exploration. Functional programming languages, such as Haskell and Erlang, have proven to be particularly suitable as host languages, and our group has used them to develop influential DSLs for software specification and testing (QuickCheck [8], hardware description and verification (Lava [7]), and generic functional programming (PolyP [22]).

Our concrete goals for this project are to *improve* the DSL-based technology we have developed, to use it to address *new and important problems*, and to combine exciting science with *actual industrial impact*. Specifically, we plan to focus our efforts in two research themes: DSL design, and Testing and verification.

Research area overview

Functional programming Functional programming—programming (mostly) without side-effects, and with powerful features such as first-class functions—dates back almost as far as high-level languages themselves. Academic interest soared after John Backus devoted his Turing Award lecture to a functional programming manifesto [6], and the area has remained active ever since. We have contributed throughout this long history, with for example an early seminal paper [19], and an important rôle in the design of Haskell, perhaps the most influential functional programming language today [18, 26].

In recent years, functional programming can truly be said to have come of age; it has become a "disruptive technology" in telecoms [1], finance [4], and many other areas. A good overview of the industrial applications of functional programming can be obtained

from the *Commercial Users of Functional Programming* workshop series, now part of the ICFP Developer Conference. With the Erlang web site serving over two million requests per month, and Microsoft's decision to market their functional language F# actively [28], the growth of "real world" functional programming seems certain to continue.

Domain specific languages Many applications of functional languages use them to define domain specific languages for particular problems. DSLs have become popular tools for solving many software development problems [29], but they fit particularly well with functional programming. The key idea is to *embed* the domain specific language within a host language as a library, so that the syntax of the DSL consists of function calls to the library in the host language, resulting in a *domain specific embedded language* (DSEL), a term coined by Hudak [17]. This simple idea enables the DSL to inherit the infrastructure of the host language, and dramatically simplifies the implementation. The idea fits well with functional programming because *binding* constructs in the DSL can be implemented via *higher-order functions* in the host language; the availability of λ-expressions in the host language is critical.

While many DSELs are essentially interpreted by the host language, this is by no means essential—DSELs can be compiled to another target language by constructing the library so that the evaluation of a DSEL program in the host language actually generates code in another. This idea was introduced by Leijen and Meijer [23], who coined the name *domain specific embedded compiler*, and further refined by Elliott et al [13].

Our own work on DSELs has followed these lines as well. Lava [7], our tool for hardware design, is a domain-specific embedded compiler with several back-ends—including for example VHDL for FPGA generation and SMV for automated model checking. Two more recent examples are Wired [5], a hardware description language with fine-grained control over layout, and Obsidian [27], a GPU programming language that provides fine-grained control over threads and memory layout. Both are high-level programming languages, but offer control over important low-level details. We have also developed a DSEL for climate impact modelling [24]. At present it is a specification language that can be used to manually create high-performance climate-impact models in C++ from high-level specifications.

Random testing Our random testing tool QuickCheck [8] is another example of a DSEL—in this case, the embedded language is a subset of predicate calculus used to define properties of a program, and the effect of executing the DSEL is to test the properties in randomly generated cases. Thus we test code directly against a formal specification. Originally developed in Haskell, the idea is so attractive that it has been emulated in many other languages. A reimplementation in Erlang is the basis for Hughes' start-up Quviq AB, whose product is used in particular in the telecoms industry [2].

Random testing in general has enjoyed a renaissance in recent years, with a dedicated workshop founded in 2006 [21]. Particularly influential is Godefroid et al's DART [15], which combines random testing with constraint solving to reach parts of the code that would otherwise be hard to cover. The combination of random testing with model-checking

techniques is found in many recent tools, such as CUTE, EXE, SMART and PEX. However, these are of necessity "white box" methods—they take the structure of the code under test into account—while our focus is rather on "black box" testing in which the structure of the code under test is unknown. (Telecoms systems are often built using a combination of programming languages, making language-specific white box methods difficult to apply).

One of QuickCheck's most valuable features is *shrinking*, which automatically reduces failed test cases to a minimal failing example. This separates the "signal" (that causes a test to fail) from the "noise" in random test data, and is crucial to speedy fault diagnosis. Shrinking is related to Hildebrandt and Zeller's *delta-debugging*, a simplification method for failing cases which has been successfully applied in many different contexts [16].

Our EU FP7 project ProTest is based to a large extent on QuickCheck; under its auspices we are developing property-based testing methods for concurrent programs.

Specification Property-based testing inherently requires a formal specification of the code under test: the properties that we test themselves comprise a (usually partial) specification. One of the difficulties of applying the idea is that specifications of real software are almost always *informal*, often ambiguous, and real software developers do not always find it easy to formalize them. This motivates *specification mining* from existing code—the dynamic discovery of specifications from test results. For object-oriented code Daikon [14] discovers *invariants* which appear to hold at internal program points—for example, that a pointer is non-null, or a linear relationship between index variables. Much less attention has been devoted to discovering the *external specification* of an API.

Automated Reasoning Tools A DSEL is often equipped with multiple back ends, some of which perform analyses whose results are fed back to the source level. Particularly useful are analyses based on automated reasoning, that answer questions about the domain specific code posed in a particular logic—for example propositional, temporal, or first-order logic. Therefore, such reasoning tools are an important element of our work on DSELs.

Our SAT-solver MiniSAT [12] is a multiple winner of the yearly world-wide SAT competition, and is today the "standard" open-source SAT-solver that researchers in the field either base their work on or compare themselves with. MiniSAT is also the basis of many of our other automated reasoning tools; notably Tip, a symbolic model checker that recently won the BMC part of the model checking competition HWMCC, and Paradox [10], a finite model finder for first-order logic, that has won the relevant category of CASC, the World Championship on Theorem Proving, every year since its inception in 2003.

Project description

Our proposal falls into two related parts: *DSL Design* and *Testing and Verification.*

DSL design Our efforts in this area comprise the design of *specific* DSLs for particular problem areas on the one hand, and the development of a DSL *toolkit* on the other.

We have already begun developing a DSL for digital signal processing (DSL4DSP), with support from Ericsson, which directly addresses the problems discussed in the introduction. This particular DSL has several apparently conflicting purposes. It must first of all be a specification language, enabling communication and algorithm design exploration—one of Ericsson's major aims is to produce algorithm descriptions that are independent of their final hardware-dependent implementations. To support this, we need analysis methods and cost models that work on DSL descriptions, enabling the rapid exploration of choices such as data layouts in memory or the use of tables. Users will later likely generate and examine C for a particular platform, but it is important also to support higher level analyses, to encourage a true raising of abstraction level. We aim to make this specification language look very like a subset of native Haskell—but generate optimised C from it, which will demand very clever implementation strategies! We must, in addition, develop methods that assist the user in thinking about and controlling the effects of caches, while bearing in mind that the language should later be extended for use in multicore programming.

The second purpose is to enable generation of optimised implementations for particular hardware, while supporting later re-partitioning. We will develop a battery of DSL compilation techniques, including new fusion methods enabling high quality code generation (IFA project application, Axelsson). We will explore the use of dynamic programming and search in mapping kernel algorithms to a DSP that provides particular opportunities for parallelism (multiple ALUs or other accelerators, VLIW) (VR project application, Sheeran). This includes developing ways to model the DSP itself, and thus builds on our previous experience in hardware description. It is vital, also, to develop new methods of relaying information from code generation back to the DSL source, so that the user can understand the relationship between the generated C and the higher level description.

The key to success in developing this kind of DSL is understanding what is best automated, and what the user should control explicitly. Instead of relying on an increasingly clever compiler to optimise a *fixed* language, we aim to *extend the DSL* to capture the critical design decisions affecting performance of the final code, combining high and low levels in one description. Case studies and real-world examples must guide us here—capturing and elucidating the necessary abstractions is an important aspect of DSL design.

We are also developing DSLs in other application areas: GPU programming (Obsidian), high level hardware modelling (with Intel), and climate impact modelling [24]. The latter is just a specification language at present; there is as yet no automated connection to the C++ codes that climate impact researchers use to evaluate their hypotheses. We aim to make this connection, extending the DSL with a back-end which can generate high-performance climate-impact models from high-level specifications.

Our experience of designing many different DSLs has shown us that similar problems arise in many DSL designs. We believe it is now time to generalise their solutions into usable libraries for building DSLs. Our previous work on Observable Sharing [9], which enables a DSL-compiler to detect the internal structure of programs, and Arrows [20], a

powerful generalization of monads, can be seen as examples of such libraries. Currently, we are capturing other common patterns we see in our DSLs as general libraries, such as: the concept of abstract "boxes" with typed inputs and outputs that can be connected (to model modular programs); the concept of "virtual" data structures (such as arrays) that only exist conceptually to the programmer, but are simplified away during program generation; and handling variable binding, with the aim of inspecting functions as first-class values. We would also like to tackle the perennial problem of generating meaningful error messages from embedded DSLs. The aim is to make a toolkit for building DSLs, as a collection of libraries that each solves a challenging problem in DSL design and implementation.

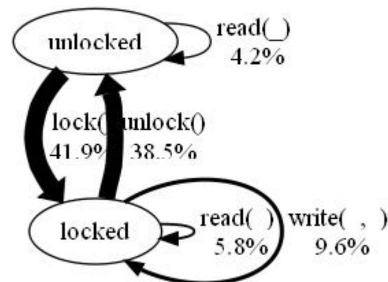
One important feature of many DSLs is that they permit more far-reaching *analysis* of domain-specific programs, than a general purpose language can support. Here, we will continue to use automated reasoning tools to reason about functional correctness properties of programs. Tools we are continuously developing for this are MiniSAT (for propositional logic), Tip (which lifts MiniSAT to temporal logic), and Paradox and Equinox (which lift MiniSAT to first-order logic). For DSLs in data-heavy domains such as DSP, we need to develop automated reasoning tools in logics in which one can easily express data manipulation. One obvious choice is to use first-order logic, but this constitutes a challenge: How can one provide quick and useful feedback to questions posed in a logic which is only semi-decidable? We have recently developed the novel concept of *approximation model* which may be used to attack precisely this situation, and we are keen to put this hypothesis to the test and develop this idea further.

The longer term goal is programming methods that enable the combination of a higher level of abstraction with the control of those fine details that need to be under the control of the programmer. This goal is made concrete by the choice of telecoms programming as an application area in which both individual DSLs and theories and methods for DSL construction can be tested on concrete, demanding case studies. Our approach is compatible with the Berkeley "view" [3], in which programming methods and auto-tuning for standard *motifs* (groups of related algorithmic problems) are developed.

Testing and verification Our goals in this part of the project are to place property-based testing on a firmer theoretical foundation, to improve the effectiveness of random test-case generation, to aid in the construction of specifications, and to evaluate the effectiveness of "property-first development".

QuickCheck can be thought of as a DSL for program properties, whose interpretation is by automatic testing. Yet a big part of its appeal is that the properties also have a logical meaning and can be seen as a formal logical specification of the program as well. Any given property can usually be expressed in many different, but logically equivalent ways. However, these may behave quite differently as automated tests—some of them might not even be testable at all! Thus, it is desirable to have a way of reasoning about not only the logical properties of a specification, but also its *testability properties*. We have begun to develop a *testing logic*, which helps in comparing the testability of logically equivalent specifications of programs, and may lead to "optimal" ways of testing a given specification.

Random testing is critically dependent on good *generation* of test data, which can require considerable ingenuity. The task is not only to fulfill the required preconditions, but also to attain a *distribution* of tests that maximises the probability (per unit of test effort) of finding a bug. For example, suppose one of a set of random tests has a probability p of revealing a bug each time it is run, while the others will always succeed—but we know nothing about *which test* may fail. We could choose to "bet" on one of the tests, and devote all of our testing effort to that one, increasing our chances of finding the bug quickly if our bet is right—but the best strategy is *provably* to distribute test effort evenly across all the available tests. Finding this best strategy in general will improve the effectiveness of random testing—but it is challenging to do so, because it depends on factors such as how likely each code fragment is to be buggy, which may depend, for example, on who wrote it! Our strategy is to define a DSL for users to express test priorities, then generate test data with an optimal distribution based on those priorities. So far, we have investigated the best weighting of transitions in a finite state machine testing library. The diagram shows the result of applying our present heuristic to a simple locker state machine, given that the `lock()` transition is prioritized ten times as highly as the others. Note that the `unlock()` transition must also be followed often—even though it is not prioritized—if tests are to contain many calls of `lock()`. This kind of interaction between different states makes weight assignment by hand very hard in larger cases. The problem is related to NP-complete problems in network flow graphs, and much work remains to be done—both in refining the optimization goal function, and in finding more efficient approximation algorithms to find high quality solutions.



Even though we are engaged in "black box" testing, there is often *some* code-centered information available—code coverage or profiling information, for example. We aim to extend QuickCheck to take advantage of such information. We already have a prototype tool which uses QuickCheck to generate a *minimal test suite* that achieves the same code coverage as a long sequence of random tests; such a test suite might be used in conjunction with random tests to ensure that parts of the code that are hard to reach with random data, are nonetheless tested fairly often. We plan to extend this idea to *adapt the distribution* of random tests, to optimize the bug detection probability as discussed above.

One of the biggest problems in applying property-based testing in practice, is that developers find it hard to formulate properties. One way to ease this is to define *domain specific specification languages* (such as the finite state machine library mentioned above). We plan to relate such DSSLs to the UML models popular within Ericsson. Note that modelling does not obviate the need for testing—indeed, Executable UML [25] is gaining ground, in part because models can be validated by testing before the software itself is built. Testing requires a *separate* specification that the model can be tested against; errors are revealed by inconsistencies between two *different* descriptions of the same thing. Together with Rogardt Heldal (Chalmers), we are planning a collaboration with Toni Siljamäki at Ericsson to

investigate QuickCheck testing of Executable UML models.

Another enticing goal is to *derive specifications automatically from the software itself.* Such specifications can yield insight into the software's behaviour, they can be used for regression testing as the software evolves, and (if they are derived by testing) may actually reveal bugs! We have developed a prototype tool which can suggest algebraic specifications for library APIs, by using QuickCheck to test all conceivable algebraic equations between terms up to a certain depth, filtering out the equations that do not hold, and ending up with a "complete" set of equations. This sounds expensive, but we use smart data structures to battle the complexity of the problem. Our initial results are very promising—for example, we automatically discover laws such as $merge\ h\ (insert\ x\ h_1) = insert\ x\ (merge\ h\ h_1)$ about heaps. But how far can we take this idea? The equations we generate today have to be universally true; can we make them more useful and more applicable by generating conditional equations? The equation generation can be seen as a way of providing feedback about a library implementation; what happens when the implementation contains bugs?

Finally, property-based testing appeals to us as scientists—because of its focus on formal specifications, and its immediate feedback when specifications are violated. We believe that *code can be developed faster and better, when properties are formulated together with, or ahead of, the code they apply to.* But how do we know? Studies of test-driven development using unit tests have yielded somewhat mixed results. We plan to carry out similar experiments to investigate this hypothesis.

Group Description, Grants and Collaboration

Our group consists currently of the four PIs (Claessen, Hughes, Jansson, Sheeran), who together supervise six doctoral students and four postdocs. We collaborate closely with Arts and Svensson at the Applied IT department; they are also funded by the ProTest project[1]. Hughes' current VR project is also closely related to the topic of ProTest and of this application[2]. Three new VR projects are sought by the co-applicants, all closely related to this proposal[3].

With the support of Sweden's Strategic Research Foundation and Ericsson Software Research, Sheeran is spending a year working on a DSL for DSP programming at Ericsson in Göteborg[4]. As one result of this, Anders Persson from Ericsson, already an active collaborator, has applied to VR for an ID-project to pursue an industrial doctorate in our

[1] ProTest: "Property-based Testing" (FP7-ICT-2007.1.2, started 2008, `www.protest-project.eu`), also involves Ericsson, the Universities of Kent, Sheffield, and Madrid, and two other smaller companies. Our group's part focusses on testing concurrent programs.

[2] "Language terms as test data for property-based testing" (VR 2009–2011), focusses on generating good random test data for telecoms protocols from the grammars contained in the RFCs that define them.

[3] Claessen: "QuickSpec: Generating Specifications from Programs" (2010–2012); Jansson: "Efficient Generic Programs and Specifications" (2010–2012); Sheeran: Context-aware program generation: an application of functional programming (2010-2012).

[4] DSL4DSP: a project funded by Ericsson and SSF; involves 3 Ericsson sites, Chalmers and ELTE University in Budapest. The intention is that the resulting language should become a de-facto standard, used by DSP vendors as well as their customers.

group, to develop a DSL for the control part of base station signal processing. Axelsson, who is currently an Ericsson-funded postdoc on the DSL4DSP project, has applied for an IFA-project to continue this work. This strong link to Ericsson gives us unparalleled expertise in the problems of real-world DSP implementation. Moreover, Hughes is part-time CEO at Quviq AB, our associated company, which sells a version of QuickCheck primarily to customers in the telecoms market; this gives us insights into the problems of testing telecoms control software, and provides a direct route for technology transfer.

We enjoy high visibility internationally: three of us are members of the IFIP Working Group WG2.8 on Functional Programming, and Jansson is a member of WG2.1 on Algorithmic Languages and Calculi. Satnam Singh (from Microsoft Research) and Carl Seger (from Intel Strategic CAD Labs) are Visiting Faculty, interacting strongly with our group.

We collaborate locally with Sands' group in language based security, and Stenström's group in multicore systems. Both groups provide valuable expertise in the area of this application. Moreover, together these three groups were selected to represent our department in the large "Chalmers Initiative on ICT" proposal recently submitted to Vinnova.

Significance

Strategic relevance is guaranteed by a close collaboration with industry—indeed, the problems we have chosen to address are strongly influenced by this collaboration.

Scientifically, we will contribute to, among others: *functional programming*—by demonstrating new, important applications; *domain specific languages*—by developing a toolkit that makes new DSELs easier to develop; *automated reasoning*—by further development of our prize-winning proof tools; *digital signal processing*—by developing a high-level language that offers both portability and performance; *test automation*—by improving the effectiveness of random testing; *automated software engineering*—by mining external specifications from systems.

Finally, the difficulty of programming multicore systems is one of the grand challenges facing computer science today; much of our DSL work addresses this challenge directly.

References

[1] J. Armstrong. A history of Erlang. In *HOPL III: Proc. of the third ACM SIGPLAN conf. on History of programming languages*, pages 6–1–6–26. ACM, 2007.

[2] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq QuickCheck. In *ERLANG '06: Proc. of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10. ACM, 2006.

[3] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] L. Augustsson, H. Mansell, and G. Sittampalam. Paradise: a two-stage DSL embedded in haskell. In *ICFP '08: Int. Conf. on Functional programming*, pages 225–228. ACM, 2008.

[5] E. Axelsson, K. Claessen, and M. Sheeran. Wired: wire-aware circuit design. In *Correct Hardware Design and Verification Methods*, volume 3725 of *LNCS*. Springer, 2005.

[6] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Int. Conf. on Functional Programming, ICFP*, pages 174–184. ACM, 1998.

[8] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00: Int. Conf. on Functional Programming*, pages 268–279. ACM, 2000.

[9] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *In Asian Computing Science Conference*, pages 62–73. Springer Verlag, 1999.

[10] K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[11] N. A. Danielsson, J. Gibbons, J. Hughes, and P. Jansson. Fast and loose reasoning is morally correct. In *POPL'06*, pages 206–217. ACM Press, 2006.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *The SAT Conference*, 2003.

[13] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comp. Prog.*, 69(1-3), 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. of the 2005 ACM SIGPLAN conf. on Programming language design and implementation*. ACM, 2005.

[16] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *ISSTA '00: Proc. of the 2000 ACM SIGSOFT int. symposium on Software testing and analysis*, pages 135–145. ACM, 2000.

[17] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proc.: Fifth Int. Conf. on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[18] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proc. of the third ACM SIGPLAN conf. on History of programming languages*. ACM, 2007.

[19] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[20] J. Hughes. Generalising monads to arrows. *Science of Computer Programming, special issue on Mathematics of Program Construction*, 37(1-3):67–112, 2000.

[21] *RT '06: Proc. of the 1st int. workshop on Random testing*. ACM, 2006.

[22] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97: Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[23] D. Leijen and E. Meijer. Domain specific embedded compilers. In *In Proc. of the 2nd Conf. on Domain-Specific Languages*, pages 109–122. ACM Press, 1999.

[24] D. Lincke, P. Jansson, M. Zalewski, and C. Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell: A DSL for computational vulnerability assessment. In *IFIP Working Conf. on Domain Specific Languages*, LNCS, 2009. In Press.

[25] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, 2002.

[26] S. Peyton Jones and J. Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*. Available from `http://www.haskell.org/definition/`, Feb. 1999.

[27] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *post-symposium proceedings of 20th International Symposium on the Implementation and Application of Functional Languages (2008), in press*, 2009.

[28] D. Syme. Why Microsoft is investing in functional programming. In S. Peyton Jones and J. Grundy, editors, *Commercial Users of Functional Programming*, 2008.

[29] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.